

This Page Is Inserted by IFW Operations
and is not a part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

**As rescanning documents *will not* correct images,
please do not report the images to the
Image Problem Mailbox.**

PATENT APPLICATION

**APPARATUS AND METHOD FOR MEMORY EFFICIENT,
PROGRAMMABLE, PATTERN MATCHING FINITE STATE MACHINE
HARDWARE**

Inventor(s): Stephen Gould, a citizen of Australia, residing at
4 Henry Street
Queens Park, NSW 2024, Australia

Robert Matthew Barrie, a citizen of Australia, residing at
5 Kiaora Road
Double Bay, NSW 2028, Australia

Darren Williams, a citizen of Australia, residing at
81 Campbell Street
Newtown, NSW 2042, Australia

Nicholas de Jong, a citizen of Australia, residing at
1001/1 Hollywood Avenue
Bondi Junction, NSW 2022, Australia

Assignee: Sensory Networks, Inc.
Level 4, 140 William Street
East Sydney, NSW 2010, Australia

Entity: Small business concern

**APPARATUS AND METHOD FOR MEMORY EFFICIENT,
PROGRAMMABLE, PATTERN MATCHING FINITE STATE MACHINE
HARDWARE**

5 CROSS-REFERENCES TO RELATED APPLICATIONS

[0001] The present application claims benefit under 35 USC 119(e) of U.S. provisional application number 60/454,398, filed on March 12, 2003, entitled “Apparatus And Method For Memory Efficient, Programmable, Pattern Matching Finite State Machine Hardware”, the content of which is incorporated herein by reference in its entirety.

10

 FIELD OF THE INVENTION

[0002] The present invention relates to integrated circuits, and more particularly to memory efficient programmable finite state machines.

15

 BACKGROUND OF THE INVENTION

[0003] Deep content inspection of network packets is driven, in large part, by the need for high performance quality-of-service (QoS) and signature-based security systems. Typically QoS systems are configured to implement intelligent management and deliver content-based services which, in turn, involve high-speed inspection of packet payloads. Likewise,
20 signature-based security services, such as intrusion detection, virus scanning, content identification, network surveillance, spam filtering, etc., involve high-speed pattern matching on network data.

[0004] The signature databases used by these services are updated on a regular basis, such as when new viruses are found, or when operating system vulnerabilities are detected. This
25 means that the device performing the pattern matching must be programmable.

[0005] As network speeds increase, QoS and signature-based security services are finding it increasingly more challenging to keep up with the demands of matching packet contents. The services therefore sacrifice content delivery or network security by being required to miss packets. Currently, fast programmable pattern matching machines are implemented
30 using finite state machines (FSM). As is known, the process of mapping a regular expression,

or signature database, to a FSM involves compiling the expression into a non-deterministic finite-state automaton (NFA), and then converting the NFA to a deterministic finite-state automaton (DFA).

5 [0006] An FSM typically starts in a given initial state, usually state zero. On receipt of each input symbol, the FSM advances to a new state determined by the current state, together with the input symbol. This operation is referred to as calculating the “next state” or “transition function” of the finite state machine. The calculation of the next state is often performed through a table lookup. The table, known as the “transition table”, is arranged so as to have the row number determined by the current state and the column number by the current input
10 symbol. Each entry in the transition table contains the value for the next state given that current state, as defined by the row, and the input symbol, as defined by the column. The transition table is commonly stored using a RAM lookup table. Data symbols received from a digital network are usually encoded as 8-bit bytes, and the number of states is determined by the complexity of the given application. The following pseudo-code illustrates the FSM
15 operation:

```
20      CURRENT_STATE = 0
      for each INPUT_SYMBOL,
          NEXT_STATE = TRANSITION_TABLE[CURRENT_STATE][INPUT_SYMBOL]
      CURRENT_STATE = NEXT_STATE
      next INPUT_SYMBOL
```

[0007] Figure 1 shows a block diagram of a conventional finite state machine 10. The current state is encoded as an m-bit binary word, and the current input symbol as a k-bit
25 binary word. These bits are concatenated together by logic block 12 to form an (m + k)-bit address to a RAM lookup table 14. RAM 14 contains the state transition table, that is, each RAM entry contains an m-bit word representing the next state given the current state and the input symbol. Look-up table 16 receives data from RAM look-up table 14 to define the action to take in each particular state. This is used to indicate terminal/accept states, etc. These
30 actions are shown as being encoded as p-bit words.

[0008] Programmable FSMs are often expensive because of the size of the memory required to store the transition table. This problem is even more pronounced for fast FSMs which are required to compute the next state within a few and fixed number of clock cycles. For example, the state machine implementation shown in Figure 1, having m-bit state vector

and k-bit symbols, requires 2^{m+k} entries of m-bit words for storing the full transition table. Additional memory is required for the output look-up table. For example, for an application servicing 1Gbps network traffic, the FSM is required to compute the next state every 8ns, for 8-bit input symbols. This poses a challenging task.

- 5 **[0009]** US Patent No. 6,167,047 describes a technique in which memory optimization is achieved through usage of stack memory allowing the state machine to repeat common sub-expressions while calculating the next state within a single clock cycle. This technique uses a large memory, and therefore limits the complexity of the FSM. This technique also suffers from the problem that the stack memory is limited.

10

BRIEF SUMMARY OF THE INVENTION

- [0010]** A programmable finite state machine (FSM), in accordance with one embodiment of the present invention includes, in part, a pair of memories, and a selecting circuit coupled to each of the memories. Upon receiving a (k+m)-bit word representative of the k-bit input
15 symbol and the m-bit current state, the first memory supplies one or more matching transition rules stored therein. The selecting circuit selects the most specific of the supplied rules. The transition rules are stored in the first memory in a ranking order of generality. The second memory is configured to receive the selected transition rule and to supply the next state of the FSM. In some embodiments, the first memory is a ternary content addressable memory and
20 the second memory is a static random access memory. Furthermore, in some embodiments, the ranking order of generality is an increasing order of generality, and in other embodiments the ranking order of generality is a decreasing order of generality.

- [0011]** In some embodiments, the FSM further includes a register configured to receive and store the k-bit input symbol and the m-bit current state and to supply the (k+m)-bit word
25 supplied to the first memory. The FSM may further include a look-up table configured to supply the FSM's output. The selection circuit may be a priority encoder or a priority arbiter. To increase the depth or the width of the FSM, several commercially available CAMs can be coupled to one another.

- [0012]** The finite state machine that is implemented in hardware has significantly reduced
30 memory requirements. The contents of both the content addressable memory and the static random memory is determined by an algorithm, in accordance with the present invention. The algorithm is an iterative algorithm which minimizes the number of terms required to

represent the next-state transition functions. The data generated by this algorithm are subsequently uploaded to the two memories.

BRIEF DESCRIPTION OF THE DRAWINGS

5 **[0013]** Figure 1 is a simplified high-level block diagram of a finite state machine, as known in the prior art.

[0014] Figure 2 is a simplified high-level block diagram of a programmable finite state machine, in accordance with one embodiment of the present invention.

10 **[0015]** Figure 3 is a simplified high-level block diagram of a programmable finite state machine, in accordance with another embodiment of the present invention.

[0016] Figures 4A-4B show an example of the operation of the finite state machine of Fig. 2, in accordance with one embodiment of the present invention.

15 **[0017]** Fig. 5 is a flow chart of steps that may be used to minimize the storage requirements of the memories of the finite state machines of Figs. 2-3, in accordance with one embodiment of the present invention.

[0018] Figures 6A-6B show various tables associated with memory storage minimization method, in accordance with one exemplary embodiment of the present invention.

20 **[0019]** Figure 7 shows various tables associated with the memory minimization process of an finite state machine, in accordance with another exemplary embodiment of the present invention.

[0020] Figure 8 is a simplified high-level block diagram of a programmable finite state machine having cascaded memories, in accordance with another embodiment of the present invention.

25 **[0021]** Figure 9 is a simplified high-level block diagram of a programmable finite state machine having cascaded memories, in accordance with yet another embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

30 **[0022]** In accordance with the present invention, a programmable finite state machine (FSM), is configured to require less memory storage capacity to perform fast pattern

matching than those known in the prior art. Figure 2 is a simplified high-level block diagram of an FSM 50, in accordance with one embodiment of the present invention. FSM 50 is shown as including, in part, a register 100 adapted to store the current state and input symbol, a ternary content addressable memory (TCAM) 101 adapted to store transition rules, a priority encoder 102 adapted to select the most specific rule from all rules which match, an SRAM 103 adapted to deliver the next state in response to the rule selected by TCAM 101, and a look-up table (LUT) 104 adapted to deliver FSM 50's output defining the action the FSM 50 performs. Priority encoder 102 receives 2^n wordlines (signals) and represents the first asserted wordline as an n-bit binary coded signal.

[0023] Figure 3 is a simplified high-level block diagram of an FSM 160, in accordance with another embodiment of the present invention. FSM 160 is similar to FSM 50 except that it includes a priority arbiter 152 in place of priority encoder 102. Priority arbiter 152 receives 2^n wordlines, zero or more of these wordlines being asserted, and generates 2^n wordlines with only zero or one asserted, with the asserted wordline corresponding to the first asserted input wordline. In other words, priority arbiter 152 deasserts all but zero or one the received wordlines.

[0024] In the following, k is the number of bits required to represent each symbol, and m is the number of bits required to represent each state within the FSM 50. Thus, the total number of states is 2^m , and up to 256 symbols may be represented if k is equal to 8. TCAM 101 contains a lookup table for transitions into each state and is adapted to benefit from a sizeable amount of memory compression. TCAM 101 may have 2^n entries of length (m + k)-bits, where n is typically smaller than (m + k). The matching TCAM word is delivered to priority encoder 102 which selects the lowest address encoded as an n-bit word. Thus entries are stored in the TCAM ordered from most specific to most general. Unless otherwise stated, in the following, all numbers are assumed to be in the decimal format.

[0025] Fig. 4A shows an exemplary embodiment of an FSM 200, in accordance with one embodiment of the present invention. Fig. 4B shows a portion of the state transition diagram 205 of FSM 200. According to this transition diagram, if the FSM is in state 1 and it receives input symbol 5, the FSMs transition to state 2, otherwise the FSM remains in state 1. In this exemplary embodiment, each state is encoded using a 4-bit binary number, and each symbol is encoded using a 3-bit binary number. Two of TCAM 202 entries are also shown, namely entries 206 and 207. As shown in Fig. 4, the current state is assumed to be binary "0001", and

the input symbol is assumed to be binary "101". Accordingly, register 201 delivers the 7-bit input binary "0001101" to TCAM 202.

[0026] TCAM 202 is shown as having two entries, namely 206 and 207 that match the data received by TCAM 202. Entry 206 includes binary data "0001101", and entry 207 includes binary data "0001xxx" where "x" represents a "don't care" bit. The entries are ordered so that the more specific rule, binary "0001101", matches first, and is therefore the rule selected by the priority encoder 203. SRAM 204 contains the next state, i.e., state 2 for this combination of state and input symbol, as shown in state transition diagram 205. If while in state 1, an input symbol other than 5 is received, the second rule is selected, and the FSM remains in state 1. Priority encoder 203 encodes the address of the first match, i.e., binary "0001101", thereby causing the retrieval of the next state binary "0010" from SRAM 204.

[0027] In accordance with the present invention, the TCAM entries are computed such that they require relatively small storage capacity and are arranged such that the more specific rules are triggered before the more general ones (as already discussed). Fig. 5 is a flow chart of steps that may be used to minimize the memory usage, in accordance with one embodiment of the present invention. The flow chart is used to generate entries for both the TCAM and the RAM of the FSM.

[0028] The memory storage optimization starts in step 501. At step 502, the TCAM is cleared and a multitude of Boolean logic functions are generated for the states to which transition may be made. The clearing of the TCAM content may be done in hardware or software. A copy of the contents of the TCAM, i.e. memory image, may be maintained for purposes of the memory optimization algorithm. This memory image may be later uploaded to the hardware. One table for each state in the FSM is constructed. The table consists of 2^{m+k} rows, one for each state/symbol pair. A one (1) is placed in the row if the symbol corresponding to that row causes a transition from the state corresponding to that row, to the state for which the table is being constructed. Otherwise a zero (0) is placed in the row.

[0029] Next, at step 503 the generated Boolean logic tables are run through a standard logic minimization procedure, such as Espresso or Quine-McCluskey, as known to those skilled in the art. This produces a list of minterms for each "next-state". An example minimization method for producing the minterms by the Quine-McCluskey method is described in the attached Appendix A.

[0030] At step 504, the minterms are counted and the tables are sorted by number of minterms. The tables with the minimum number of minterms have their minterms added to the TCAM, and the corresponding state added to the SRAM, as shown in step 505. All remaining tables are modified by placing a “don’t care” in the rows corresponding to the minterms added to the TCAM, as shown in step 506. The priority encoder is configured to select the upper entries in the TCAM first. The procedure is repeated until all states have been added to the TCAM, as shown in step 507. The memory storage optimization ends in step 508. In accordance with the algorithm described in connection with flow chart 500, the last entry in the ternary TCAM consists of all “don’t cares”. This implements an “else” clause that generates a default transition if no other rules have matched. Another way to implement the “else” clause is to leave the last entry out of the TCAM, and rely on the TCAM hardware to report a “no match” condition.

[0031] Figures 6A-6B show various exemplary tables associated with memory (e.g., TCAM 101 and SRAM 103 shown in Fig. 2) storage minimization method of an FSM, such as FSM 50, in accordance with one exemplary embodiment of the present invention. State transition diagram 601 of such an embodiment of FSM 50 is adapted to detect the words steve, matt, nick, or daz regardless of whether they appear in uppercase or lowercase. Transition table for this FSM is shown in table 602 in which the rows represent current state, and the columns represent input symbol. State 13 is a terminal accepting state, and after the FSM enters this state, it remains in this state. For simplicity and clarity, the transition diagram 601 does not show some of the transitions, such as transitions back to the state 0 (zero-th state), many of the transitions to states 1, 5, 8, and 11, and the transitions from state 13 back to state 13. As described further below, a conventional FSM may encode this transition table using 182 table entries, since there are 14 states and 13 symbols (including the “else” symbol). However, using the present invention, the FSM requires only 18 entries, i.e., 18 memory locations in the TCAM 101 and 18 memory locations in SRAM 103.

[0032] Each of TCAM 101 and SRAM 103 of FSM 50 associated with the example shown in Figs. 6A-6B, require 18 entries. Table 603 shows the entries for TCAM 101, and table 613 shows the entries for SRAM 103. Using ASCII character encoding shown in table 604, the truth tables are formed for each “next-state” function. The truth tables for the first round of optimization for state 0 is shown in table 605, and the truth tables for the first round of optimization for state 1 is shown in table 606. Logic minimization on the truth table for state 1 results in one TCAM entry, shown as entry 603_1 in table 603. Logic minimization for

state 0 results in 31 TCAM entries for the first round, and that is shown in table 607 in Fig. 6B. After all the transitions--except those associated with states 0 and 13--are added to the TCAM, the logic minimization procedure results in 23 TCAM entries associated with state 0, shown in table 608. After all the transitions associated with state 13 is added to the TCAM, shown as entry 603_17 in table 603, the minimization of transitions associated with state 0 results in a single TCAM entry, shown in table 609 and also shown as entry 603_18 in table 603, thereby resulting in the implementation of an "else" clause.

[0033] The following is a pseudo-code of memory optimization, in accordance with the present invention. The memory optimization procedure may be terminated early if all remaining minterms can fit into the available memory. In this case, the remaining minterms may simply be added to the TCAM and RAM as if they had been optimized.

```

Clear the TCAM and RAM memory images
Let F to be an empty set of boolean logic functions
15 For each state,  $S$ ,
    Define the boolean logic function,  $f_S$ , for transitions to state
    Add  $f_S$  to the set of logic function F

Repeat
20   For each boolean logic function in F
       Minimise the logic function  $f_S$ 
       Count the number of minterms in  $f_S$ 
       For the logic function,  $f_{S'}$ , with the least minterms
       Remove  $f_{S'}$  from F
25   For each minterm,  $M$ , in  $f_{S'}$ 
       Add  $M$  to the TCAM
       Add  $S'$  to the RAM
       For all  $f_S$  remaining in F
       Add "don't care" terms for  $M$ 
30 Until all F is empty

```

[0034] As described above, the Boolean logic functions, f_S , are functions of the bit representations of both the current state and input symbol, that is, f_S is a Boolean function taking $m+k$ Boolean arguments, where m is the number of bits needed to represent the states, and k is the number of bits needed to represent the symbol. It is understood that state numbering is arbitrary, and renumbering of the states may affect the memory optimization process since each table entry is formed by concatenation of the state and symbol bits. Thus more compact representation of the state machine may be generated if a correlation exists between some of the states and symbols causing transitions to those states.

[0035] Figure 7 shows various tables associated with memory minimization process of an FSM, in accordance with another exemplary embodiment of the present invention. The state transition diagram 1000 of this FSM is described by the following regular expression:

5 $b^*(ac|d)c^*$

[0036] Table 1010 shows the transition table of this FSM. These states may be encoded in binary as 00, 01, 10, and 11 for states 0, 1, 2, and 3 respectively. Likewise, symbols a, b, c, and d may be encoded as 00, 01, 10, and 11, respectively. First, the sum-of-product functions for each next-state is formed as is shown in table 1020. Next, each of these functions is minimized using a logic minimization algorithm such as the well-known Quine-McCluskey described in the attached exhibit A. Such logic minimization results in the function for state 0 being represented by three terms, namely terms 001x, x10x, x1x1, the function for state 1 being represented by one term, namely term x000, the function for state 1 being represented by one term, namely term x001, and the function for state 3 being represented by two terms, namely terms x110, and 101x.

[0037] Since each next-state function cannot be represented by less than one minterm, functions for states 1 and 2 are added to the TCAM at this point of optimization. The corresponding entries in the TCAM and RAM for these two states are shown in tables 1030, and 1035 respectively. The memory optimization algorithm continues with another round of logic minimization for the remaining two functions representing states 0 and 3, and where “don’t cares” have been included for the entries already added to the TCAM. Table 1040 shows the result of this round of logic minimization.

[0038] Although in this simple example the number of minterms required by each function has not changed, the minterms at this stage of optimization are more general (i.e. there are more “don’t cares” in the minterms). Because at this stage the terms representing state 3 have fewer terms than state 0, the terms for function representing state 3 are entered into the TCAM; these are shown as entries 1050_1, and 1055_2 in the associated TCAM table 1050. The corresponding entries 1050_3, and 1055_4 to the RAM are shown in table 1055. Since, at this stage only the function representing state 0 remains, this function is represented by all “don’t care” bits, as shown in table 1060. This last function is added to the TCAM memory and is shown as entry 1070_5 in table 5. The corresponding entry in the SRAM is shown as

entry 1075_5 in table 1075. Therefore, in accordance with the present invention, each of the TCAM and RAM require 5 entries. In contrast, a conventional FSM requires 16 entries in its RAM.

[0039] The following Table I shows the required memory storage for each of TCAM and SRAM of a conventional FSM vs. those required for an FSM in accordance with the present invention.

Table I

	TCAM Memory bits	SRAM Memory bits
Conventional	0	$2^{m+k} m$
Present invention	$2^n (m + k)$	$2^n m$

[0040] Because in typical signature databases, n is smaller than $(m + k)$ typically, the memory compression described above, in accordance with the present invention, enables operation using smaller storage capacity than those known in the prior art. Thus, using the present invention, the hardware required to implement a given signature database is significantly reduced. It is understood that n must be at least as large as m , otherwise not all states can be represented.

[0041] As an example, a conventional state machine for signature-based security services, operating on a 1Gbps network, using 8-bit symbols and requiring 20-bits for encoding the states, requires 5120Mb of memory. However, for the same example, the state machine in accordance with the present invention may require as few as 112Mb and 80Mb of SRAM, thus the present invention offers a significant saving. The saving achieved would vary for each state machine, according to the operation of the invention.

[0042] Figure 8 is a simplified high-level block diagram of a programmable finite state machine 800 having cascaded TCAMs, in accordance with another embodiment of the present invention. TCAMs 701 and 702 are cascaded to form a finite state machine having a relatively deeper structure. This enables using several relatively smaller commercially available discrete TCAM chips. FSM 800 includes two TCAMs 701, and 702 and two priority encoders 703, 704. Each of TCAMs 701 and 702 is configured to receive the $m+k$ word generated by register 700. Register 700 combines the m -bits of the present state, with the k -bits of the input symbol to generate this $m+k$ word.

[0043] The n-bit outputs of the priority encoders 703 and 704 are supplied to a MUX 705. The MUX select line is connected to the “match” line of the first TCAM, [701]. Thus if a match is found in the first TCAM, the output of the MUX will be the address of this match, otherwise the output of the MUX will be the priority encoded output of TCAM 702. The output of the MUX and select line are concatenated by logic unit 706 to form an (n+1)-bit word shown in Fig. 8. A match signaled is asserted if either of the TCAMs generate a match. It is understood that more than any number of TCAMs may be cascaded to form a deep FSM, in accordance with the present invention.

[0044] Figure 9 is a simplified high-level block diagram of a programmable finite state machine 850 having cascaded TCAMs, in accordance with another embodiment of the present invention TCAMs 751 and 752 are cascaded to form a wider structures. This enables using several relatively smaller commercially available discrete TCAMs chips that have smaller word widths. As seen from Fig. 9, each of TCAMs 751 and 752 receives half of the (m+k)-bit word, where the current state is represented by m-bits and the current input symbol is represented by k bits. The outputs of the TCAMs are applied to AND gate 753. The output of AND gate 753 is, in turn, applied to priority encoder. It is understood that more than any number of TCAMs may be cascaded to form a wide FSM, in accordance with the present invention.

[0045] An FSM, in accordance with the present invention, may be used for, e.g., performing high-speed regular expression pattern matching on network data. Such a pattern matching is applicable in such areas as network intrusion detection, network monitoring and surveillance, virus protection, traffic filtering, content and copyright classification, storage area network indexing, policy and access control auditing, spam detection and prevention, fraud detection and network forensics, content-aware switching, message classification, etc.

[0046] The above embodiments of the present disclosure are illustrative and not limitative. The invention is neither limited by the size of the input symbol nor by the number of states to which the finite state machine may transition to. The invention is not limited by the type of memory used by the finite state machine. The invention is not limited by the type of algorithm used to minimize the number of minterms for each of the Boolean logic functions. Other additions, subtractions, deletions, and modifications may be made without departing from the scope of the present invention as set forth in the appended claims.